(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 May 2005 (12.05.2005)

PCT

(10) International Publication Number
WO 2005/043327 A2

(51) International Patent Classification[7]: G06F

(21) International Application Number:
PCT/US2004/036054

(22) International Filing Date: 29 October 2004 (29.10.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/516,037 30 October 2003 (30.10.2003) US

(71) Applicant *(for all designated States except US)*: **DOCOMO COMMUNICATIONS LABORATORIES USA, INC.** [US/US]; DoCoMo Communications Laboratories USA, Inc., 181 Metro Drive, Suite 300, San Jose, CA 95110 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **ZHOU, Dong** [CN/US]; 1470 Cedarmeadow Court, San Jose, CA 95131 (US). **ISMAEL, Ali** [US/US]; 734 Brentwood Court, Los Altos, CA 94024 (US). **SONG, Yu** [CN/US]; 1346 Elkwood Drive, Milpitas, CA 95035 (US). **ISLAM, Nayeem** [US/US]; 3370 Coak Oak Way, Palo Alto, CA 94303 (US).

(74) Agent: **KWOK, Edward**; MacPherson Kwok Chen & Heid LLP, 1762 Technology Drive, Suite 226, San Jose, CA 95110 (US).

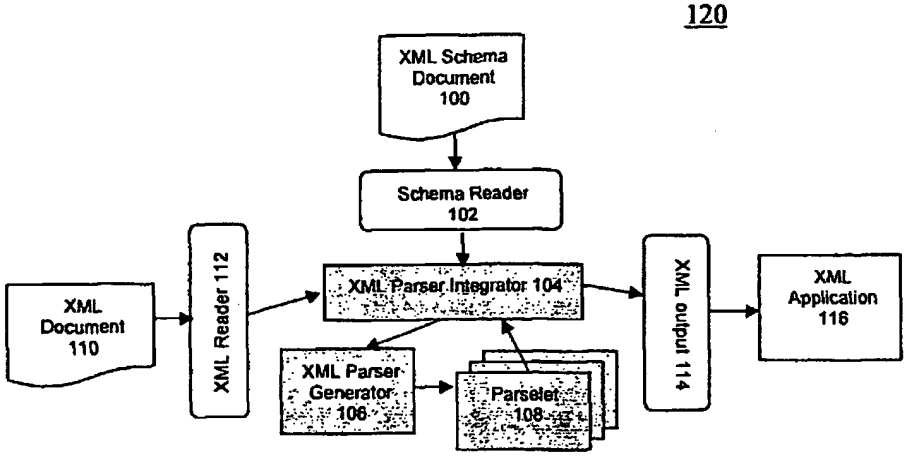
(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: METHOD AND APPARATUS FOR SCHEMA-DRIVEN XML PARSING OPTIMIZATION



(57) Abstract: Schema-driven XML parsing techniques allow an XML parser to optimize its parsing process by composing parse and to dynamically generate parsing code components based on XML schema definition for the targeted XML document. These techniques reduce the XML parsing time and reduce the memory requirement during parsing process. Further, a reconfigurable parser is provided which is guided during parsing of the XML document by XML element lexicographical information and state transition information extracted from a schema associated with the XML document. Pre-allocated element object pools may be provided based on the schema analysis to reduce the requirements for dynamic memory allocation and de-allocation operations.

WO 2005/043327 A2

METHOD AND APPARATUS FOR SCHEMA-DRIVEN XML PARSING OPTIMIZATION

CROSS REFERENCE TO RELATED APPLICATIONS

The present application claims priority to U.S. provisional application no. 60/516,037 filed on October 30, 2003, incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to methods and systems for data exchange in an information processing system. In particular, the present invention relates to providing an optimized parser for processing a structured document (e.g., a XML document) in an information processing system.

2. Discussion of the Related Art

XML is a platform-independent text-based document format[1] designed to be used in structured documents maintained in an information processing system. XML documents (e.g., forms) have become the favored mechanism for data exchange among application programs sharing data over a network (e.g., the Internet). XML documents have the advantages that (a) the information in an XML document is extensible (i.e., an application program developer can define a document structure using, for example, an XML schema description), and (b) through the XML schema, an application developer can control the range of values that can be accepted for any of the XML element or attribute in the structured document. For example, in an XML schema-defined form for a pair of shoes, the application program developer may constrain the shoe size attribute accepted by the form to be between 5 and 12. As a result, the form would reject as invalid input a shoe size of 100.

Because of these advantages, XML is widely used in consumer application programs. However, additional processing overhead is imposed on the application program to allow XML to be read and edited easily by a human using a word processor as interface, because the structured data in an XML document are required to be parsed by the application into representations that can be manipulated in the computer by the application program. Parsing requires intensive computational resources, such as CPU cycles and memory bandwidth, as

[1] In this description, the platform-independent text-based document format means a text format for defining a document which is independent of the underlying software platform (e.g., the operating system), the underlying hardware platform, or both.

the application program processes the XML elements or attributes one character at a time, in addition to implementing the higher level processing requirements of the XML schema. In a typical XML document, there can be a large number of elements and attributes which are defined in the schema using different data types and constraints. Character-matching is not efficient in existing hardware implementations, such as those based on IA32 and ARM architectures.

Parsers for documents written in numerous languages have been developed and used throughout the history of computers. For example, the first widely accepted parsers (which also validate) for XML are based on the W3C Document Object Model (DOM). DOM renders the information on an XML document into a tree structure. Thus, a parser based on DOM constructs a "DOM tree" in memory to represent the XML document, as it reads the XML document. The DOM tree is then passed to the application program which traverses the DOM tree to extract its required information. Constructing a DOM tree in memory is not only time-consuming, it requires a large amount of memory. In fact, the memory occupied by a DOM tree is usually 5 – 10 times greater than that of occupied by the original XML document. One optimization constructs a partial DOM tree in memory as needed to reduce the memory requirement and the processing time.

Alternatively, an XML document may be parsed based on a streaming model. Parsers using the streaming model include SAX and Pull. Under the streaming model, rather than a parse tree, a parser outputs a continuous stream of XML elements, together with the values of their attributes, as the XML document is parsed. Typically, such a parser reads from the XML document one XML element at a time, and passes to the consuming application the values of the element and their associated attributes. Although a streaming-based parser is efficient in its memory and processing speed requirements, such a parser merely tokenizes a string into segments of text without interpretation. The interpretation of data contained in each text segment is entirely left to the consuming application program. Thus, the burden of XML processing -- which is to provide data in an XML document to the application program in a manner that can be readily used by the application program -- is shifted from the parser to the consuming application program.

A parser may or may not validate an XML document. Validation is the process by which each parsed XML element is compared against its definition defined in an XML schema (e.g., an XML DTD file). Validation typically requires string pattern-matching as the validation program searches the multiple element definitions in the XML schema. A conventional approach to simplify validation is to convert the definitions of an XML schema into component models, expressed as a series of Java bean classes. An application program may then check the XML elements using methods provided in the Java bean classes. While

schema conversion methods may speed up both the parsing and the validation processes to some limited degree, such conversions do not provide the fast string pattern-matching desired in XML parsing and validating.

As is apparent from the above, XML parsing involves a substantial amount of string-matching operations, which are the most CPU intensive operations in XML parsing. Further, the memory requirements of parsing XML elements also lead to a substantial amount of inefficient memory allocation and de-allocation operations.

Summary

The present invention provides an XML validating parser that can dynamically generate executable parsing codes based on information extracted from an XML schema document that is either stored locally or obtained from a remote machine via network. According to another aspect of the invention, a schema-based, reconfigurable parser is provided.

In one embodiment, each XML element of an XML document is parsed and validated using a dedicated executable parsing code ("parselet"), which navigates the structure of the XML element, its attribute values and constraints to validate the element. If the element is valid, the examined XML element is passed to a consuming application program requiring the XML document to be processed. Otherwise, an invalid exception is raised and the consuming application program is notified. Because parsing in this instance is performed in a compiled executable parselet, parsing is faster than the interpretive parsers of the prior art and the memory requirement for string matching can be much reduced.

According to one embodiment of the present invention, a lexicographical analysis of the XML elements is performed in advance for a given XML schema to provide: (1) state-transition sequence information, and (2) element and attribute lexicographical distance information. The transition sequence information can be used to guide the parser as to the XML elements that may be expected to follow according to the given schema. The element and attribute lexicographical distance measures a minimal lexicographical distance between two strings (i.e., the smallest indices in the strings sufficient to identify and distinguish the strings). This information is useful for guiding the parser to identify the element or attribute of the XML document using the minimal amount of string comparison.

In one embodiment of the present invention, pools of XML element objects having pre-determined element-attribute structures are created when the parser is instantiated, which are dynamically managed so that the sizes of the pools vary as needed. A schema analysis method provides memory requirement information to allow a parse tree of the XML

document be built in memory as a DOM object using objects in the element object pools . The schema analysis also provides information for managing the sizes of element pools and the type of element objects in each pool. Having element object designs in the pools alleviate the parser's memory management requirements.

A parselet of the present invention is a compiled, executable code that executes much faster than a corresponding interpretive code which examines the schema tree and XML document tree at run time. In addition, the present invention provides multiple parselets for different XML elements, so that parallel processing of multiple XML elements simultaneously is possible. The compiled parselets may be used for multiple XML documents based on the same XML schema.

An XML schema-driven parser of the present invention may be configurable to output XML elements in the form of a DOM tree (or a similar parse tree), or in a stream, as appropriate, according to the consuming application program. The parse tree of the present invention need not provide in memory an entire DOM tree. The present invention allows a partial parse tree to be constructed on demand, including as little as a single element. Thus, the memory requirement may be reduced significantly while still allowing application programs to access XML document using DOM APIs.

By using minimal lexicographical distances between XML elements, string-matching operations are significantly reduced, thus significantly reducing the parser's demand on computational power.

By using pools of XML elements, significant amount of dynamic memory allocation operations may be avoided during parsing. When an element object is no longer needed (e.g. in the case of a streaming-based parsing), the element object is returned to the respective pool to be reused instead of being de-allocated. As a result of maintaining element pools, significant reductions in the requirements on CPU and memory resources are achieved. For example, the need for a garbage collector process -- which does not reclaim memory from finalized objects immediately -- may be avoided. Avoiding the need for a garbage collection process also reduces the requirements on the CPU.

The present invention is better understood upon consideration of the detailed description below and the accompanying drawings.

Brief Description of the Drawings

Figure 1 is a block diagram of schema-driven parser generator system 120, according to one embodiment of the present invention.

Figure 2 is a flow chart illustrating the operations of schema-driven parser generator system 120 under push mode.

Figure 3 summarizes the operations of schema-driven parser generator system 120 under push mode.

Figure 4 is a flow chart illustrating the operations of schema-driven parser generator system 120 under pull mode.

Figure 5 summarizes the operations of schema-driven parser generator system 120 under pull mode.

Figure 6 shows dynamically reconfigurable parser system 620, according to another aspect of the present invention.

Detailed description of the Preferred Embodiments

The present invention provides a validating parser that validates an XML document as it is parsed, thereby reducing the validating and parsing time requirements and the memory bandwidth requirement. The present invention may be applied to implement a parser code generator and a reconfigurable parser.

According to one embodiment of the present invention, based on a specific XML schema, a validating parser generator dynamically generates an executable parser code ("parselet") for implementing parsing of a specific XML element in the XML schema. Figure 1 is a block diagram of schema-driven parser generator system 120, according to one embodiment of the present invention. As shown in Figure 1, schema-driven validating parser system 120 includes:

(1) schema reader 102, which reads one ore more XML schema documents (e.g., XML document 100);

(2) XML reader 112, which reads XML documents (e.g., XML document 110);

(3) XML parser integrator 104, which coordinates among the different components of the XML parser (in this example, the XML parser consists of (a) XML parser integrator 104, (b) XML parser generator 106, and (c) parselets (e.g., parselet 108); and

(4) XML output module 114, which outputs validated, parsed XML elements of the XML document for use by XML application program 116.

Parselets (e.g., parselet 108) are excutable parsing codes created by XML parser generator 106 based on XML elements read by schema reader 102 from XML schema document 100. Each parselet is called to parse an XML element in an XML document (e.g., XML document 110) and to extract element and attribute values from the XML element and all its included XML elements. The parsed elements are output by XML output module 114 to XML application program 116. Schema-driven XML parser system 120 is especially beneficial when multiple XML documents are based on the same underlying XML schema, so that each parselet can be re-used multiple times.

XML schema document 100 may be a document retrieved from a local storage or from a remote storage via a network (e.g., the Internet) using one of various network transport protocols (e.g., FTP, HTTP, and SSL). Schema reader 102 reads the XML elements from XML schema document 100 one element at a time. In this embodiment, if an element is nested (i.e., it contains other XML elements), the contained elements are read before reading of the containing element is complete. Schema reader 102 may implement a push style or a pull style of reading XML elements. Under a push style, schema reader 102 continuously read XML schema document 100 until an entire element is read, whereupon schema reader 102 notifies XML parser integrator 104. Under a pull style, XML parser integrator 104 requests that schema reader 102 read the next XML element from schema document 100.

When an entire element is completely read, XML parser integrator 104 causes XML parser generator 106 to generate a corresponding parselet for the XML element read. XML parser integrator 104 maintains a mapping table, which includes all relationships between a parselet, all its containing parselets and all the parselets it contains. In addition, the mapping table also records a name space and a qualified name for each parselet (a qualified name is typically a prefix encoding a path to the parselet).

To illustrate an application of the present invention, an example of an XML document is provided in a "PurchaseOrder" document shown in Appendix A. As shown in Appendix A, PurchaseOrder is an XML element which includes other XML elements "shipTo", "billTo", "comment" and "items". Elements "shipTo" and "billTo" each include instances of elements "name", "street", "city", "state" and "zip". Element "items" may include one or more instances of element "item." Element "item" may include instances of elements "productName", "quantity" "USPrice", "comment" and "shipDate". One or more attributes may be found in each element, which values are provided by a string representing the appropriate data type. For example, element "PurchaseOrder" includes attribute "orderDate" and element "shipTo" includes attribute "country". Appendix A is the form of the document that is typically exchanged between the client (e.g., a web browser) and the application program. The corresponding schema document is shown in Appendix B.

Appendix B shows a schema which includes, at the top level, elements "purchaseOrder" and "comment". Element "comment" is defined in the schema to be a string. Element "purchaserOrder" is defined in the schema to be an element of the data type "PurchaseOrderType", which is defined to include elements, sequentially, "shipTo", "billTo", "comment" and "items", and attribute "orderDate" as already seen in Appendix A. The term "sequence" indicates the order in which the elements appears in the schema is expected to be the order in which those elements appear in the XML document. The schema further defines that the elements "shipTo" and "billTo" are both of the data type "USAddress", which is defined to include elements "name", "street", "city", "state" and "zip", as also seen in Appendix A. The schema defines "name", "street", "city" and "state" each to be a string and "zip" to be of the data type "Decimal". Similarly, element "items" is defined in the schema to include zero or more instances of element "item". Element "item" includes, sequentially, elements "productName", "quantity", "USPrice", "comment" and "shipDate". Element "item" also has an attribute "partNum" which is of the data type "SKU" -- a format for a part number which is also defined in the schema.. The data type of element "item" is not provided a name. Using the information specified in the schema of Appendix B, parser generator 106 generates the parse code for parsing the elements as they appear in the XML document. One example of the parse code for element "purchaseOrder" is shown in Appendix C.

Thus, XML parser generator 106 generates a parselet for the element "purchaseOrder" which includes also code generated for parsing all the included elements. Note that, in this embodiment, as the simple data types "string", "Decimal" and "Integer" and the various variations of the data type "Date" are encountered frequently in XML documents, the parse code for these data types are not generated specifically for every schema. Rather a base class "Parselet" is provided, and the specifically generated parselets, such as "purchaseOrder" is derived from the "Parselet" class, so that parse codes for these common data types are associated with every specifically generated parselet. An example of "Parselet" class is provided in Appendix D.

In Appendix D, the methods for parsing data types "string", "Decimal" and "Integer" and the various variations of "Date" are provided as "parseString", "parseDecimal" and "parseInteger" and "parseDate", respectively. In addition, methods for validating elements and attributes (e.g., "isElement" and "isAttribute") are also provided in class "Parselet". During parsing, "Parselet" keeps track of its progress through the XML document – i.e., where in the XML document is the current text object being parsed -- by the method "EEMoveCursor". Error condition or "exception" handler "InvalidSchema" may be called from "Parselet". An example of "InvalidSchema" is provided in Appendix E. As discussed above, the output from the parsing operation may be a DOM tree, which is built from a

number of DOM nodes interconnected from a root node. An example of some pseudocode for creating the DOM tree is provided the class "Node" in Appendix F.

Returning to Appendix C (i.e., the listing of generated parselet "purchaseOrder"), according to the structure of the XML document "purchaseOrder" as defined in the XML schema, parselet "purchaseOrder" parses both elements "purchaseOrder" and "comment" at the top level of the schema. To parse element "purchaseOrder", the method "parsePurchaseOrderType" is called to handle the data type "purchaseOrderType". Method "parsePurchaseOrderType" parses, sequentially, the required attribute "OrderDate", and each of elements "shipTo", "billTo", "comment" and "items". As elements "shipTo" and "billTo" are both of the same data type "USAddress", parsing of each element is handled by method "parseUSAddress". Element "items" is handled by method "parseItems", which is also generated according to the structure defined in the schema. As the data type of element "item" contained in element "items" is not given a name, XML parser generator 106 gives the method for handling this data type the name "parseUnnamed1". Method "parseUnnamed1" parses, sequentially, the required attribute "partNum" and elements "productName", "quantity", "USPrice", "comment" and "shipDate". Note that the parsing code also validates each element including testing if the supplied values are within the accepted range of values for each element. Attrribute "partNum" is parsed using the generated method "parseSKU". As each element is successfully parsed, a node corresponding to the element is added to the parse tree using method "addChild" in the class "Node" defined in Appendix F.

During actual parsing operation, when an element is read by XML reader 112 from XML document 110, a corresponding parselet (say, parselet 108) is selected from mapping table, based on the name space, the element's qualified name, and the relationships involving parselet 108. When parselet 108 completes its parsing task, the parsed XML element is forwarded to parser integrator 104, which passes the parsed element to XML output module 114.

According to one embodiment, parsed XML elements output from the parselets are validated against the elements' definitions in their respective schemas. Here, the term "parsed" may mean either (1) that the XML element has been converted into a structured data representation, such as a DOM tree, or (2) that the textual XML element has been validated by parselet 108 to conform its corresponding definition in the schema document. In the case of a DOM tree, an application program can directly access the XML element through XML output module 114. Alternatively, i.e., the validation of the textual XML element without more requires both lesser processing time and memory, relative to building a DOM tree.

Figure 2 is a flow chart illustrating the operations of schema-driven parser generator system 120 under push mode. As shown in Figure 2, at step 200, XML reader 112 reads

XML document 110 one element at a time. When XML element is completely read, XML reader 112 generates an event, which is ssigned a sequence number at step 202. The sequence number may be provided in ascending order by a counter. At step 204, XML reader 112 notifies XML parser integrator 104 of the event. Upon notification of the event, at step 206, XML parser integrator 104 examines and uses the name space and qualified name of the XML element associated with the event to select from the mapping table an appropriate parselet. At step 208, the selected parselet parses and validates the XML element. The parselet may provide its output in a DOM tree structure, or simply provide a textual representation of the XML element, according to the requirement of XML application program 116. At step 210, the selected parselet notifies XML parser integrator 104 of the parsed XML element. XML parser integrator 104 then pass the result of the parsing, together with information regarding the event, to XML output module 114 at step 212. XML output module 114 maintains a queue of parsed elements sequentially of event sequence numbers. At step 214, XML output module 114 notifies application program 116 of the parsed element being added to the queue.

Figure 3 summarizes the operations of schema-driven parser generator system 120 under push mode.

Figure 4 is a flow chart illustrating the operations of schema-driven parser generator system 120 under pull mode. As shown in Figure 4, in the pull mode, XML application program 116 initiates the parsing and validating process by asking the XML output module 114 for the next XML element at step 300. XML output module 114 in turn requests the next XML element from XML parser integrator 104 at step 302. At step 304, XML parser integrator 104 then requests XML reader 112 read the next XML element from XML document 110, which is accomplished at step 306. At step 308, XML reader 112 passes the XML element read to XML parser integrator 104, which selects at step 310 corresponding parselet 108 for validation and parsing based on the XML element's name space and qualified name. At step 312, selected parselet 108 then parses and validates the XML element. As in the push mode, the parsed XML element may be represented in a DOM tree structure, or in a textual representation, according to the requirements of application program 116. Parselet 108 provides the parsed XML element to XML parser integrator 104 at step 314. At step 316, XML parser integrator 104 provides the parsed XML element to XML output module 114, which provides the parsed XML element to XML application program 116 at step 318.

Figure 5 summarizes the operations of schema-driven parser generator system 120 under pull mode.

According to another aspect of the present invention, Figure 6 shows dynamically reconfigurable parser system 620. As shown in Figure 6, reconfigurable XML parser 603

uses schema analyzer 601 to obtain in lexicographical order XML elements defined in a schema document 600. Reconfigurable parser 603 then compares between every pair of adjacent XML elements to determine a minimal lexicographical distance between them. These minimal lexicographical distances guide reconfigurable parser 603 to identify XML elements during parsing. That is, reconfigurable parser 603 need only perform pattern-match sufficient to recognize the attribute or element being parsed. In addition to minimal lexicographical distances, schema analyzer 601 also provides state-transition information, such as a list of possible next elements that may appear in the XML document, as determined based on the current state.

Based on the information provided by schema analyzer 601, reconfigurable parser 603 parses XML document 604. According the present invention, reconfigurable parser 603 manages a number of element pools 605, which are created at system initialization according to the expected elements to be encountered. Each element pool includes a number of pre-allocated data structure ("XML element object") created from a template of an expected XML element. As each element is successfully parsed, an XML element object is retrieved from the appropriate element pool and assigned as a node in a parse tree. Element pools 605 are resizable and can vary in size dynamically, as reconfigurable parser 603 parses XML document 604, according to the size and complexity of XML document 604.

In one embodiment, application program 602 invokes XML parser 603 to parse XML document 604. Initially, reconfigurable parser 603 identifies the references in XML document 604 to XML elements defined in XML schema document 600. The schema references are provided schema analyzer 601 to retrieve previously extracted lexicographical and state-transition information corresponding to these references. Reconfigurable parser 603 then parses the XML references, requests XML element objects corresponding to the parsed elements from XML element pools 605, fills the XML element objects returned with the parsed data, links the XML element object to a parsed tree. When all XML references are parsed, the parse tree is provided to application program 602.

The above detailed description is provided to illustrate the specific embodiments of the present invention and is not intended to be limiting. Numerous modifications and variations within the present invention are possible. The present invention is set forth in the following claims.

Appendix A

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
        <item partNum="872-AA">
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

Appendix B

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

 <xsd:element name="comment" type="xsd:string"/>

 <xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress"/>
   <xsd:element name="billTo" type="USAddress"/>
   <xsd:element ref="comment" minOccurs="0"/>
   <xsd:element name="items"  type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>

 <xsd:complexType name="USAddress">
  <xsd:sequence>
   <xsd:element name="name"   type="xsd:string"/>
   <xsd:element name="street" type="xsd:string"/>
   <xsd:element name="city"   type="xsd:string"/>
   <xsd:element name="state"  type="xsd:string"/>
   <xsd:element name="zip"    type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
     fixed="US"/>
 </xsd:complexType>

 <xsd:complexType name="Items">
  <xsd:sequence>
   <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity">
       <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
         <xsd:maxExclusive value="100"/>
        </xsd:restriction>
       </xsd:simpleType>
      </xsd:element>
      <xsd:element name="USPrice"  type="xsd:decimal"/>
      <xsd:element ref="comment"   minOccurs="0"/>
      <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="partNum" type="SKU" use="required"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>

 <!-- Stock Keeping Unit, a code for identifying products -->
 <xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
   <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
 </xsd:simpleType>

</xsd:schema>
```

Appendix C

```
/*
 * Created on Aug 17, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package com.docomo.ss.examples;

import com.docomo.ss.InvalidSchema;
import com.docomo.ss.Node;
import com.docomo.ss.Parselet;

/**
 * @author zhou
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
public class PurchaseOrder extends Parselet {
      private static char[] ename_purchaseOrder = {'p', 'u', 'r', 'c', 'h',
'a', 's', 'e', 'O', 'r', 'd',
                  'e', 'r'};
      private static char[] ename_comment = {'c', 'o', 'm', 'm', 'e', 'n',
't'};

      public Node parse (char[] d, int off) {
            doc = d;
            offset = off;

            Node res = Node.create ();
            int i = offset++;
            while (true) {
                  try {
                        if (doc[i] == 'p' &&
isElementFrom1(ename_purchaseOrder)) {
                              increaseOffset (ename_purchaseOrder);

      res.addChild(parsePurchaseOrderType("purchaseOrder"));
                        } else if (doc[i] == 'c' &&
isElementFrom1(ename_comment)) {
                              increaseOffset (ename_comment);
                              res.addChild(parseString(ename_comment,
"comment"));
                        } else throw new InvalidSchema ();
                        // try to check validity and exit condition here
                  } catch (Exception e) {
                        e.printStackTrace ();
                        break;
                  }
            }
            return res;
      }

      private static char[] ename_shipTo = {'s', 'h', 'i', 'p', 'T', 'o'};
      private static char[] ename_billTo = {'b', 'i', 'l', 'l', 'T', 'o'};
      private static char[] ename_items = {'i', 't', 'e', 'm', 's'};
     private static char[] aname_orderDate = {'o', 'r', 'd', 'e', 'r', 'D',
'a', 't', 'e'};
```

```
Node parsePurchaseOrderType (String name) throws InvalidSchema {
      Node res = Node.create ();

      // Parse attributes
      if (isAttribute (aname_orderDate)) {
            increaseOffset (aname_orderDate, 2);
            res.addAttribute (parseDate (), "orderDate");
      }

      while (doc[offset++] != '<');

      // parse a sequence of elements
      if (isElement (ename_shipTo)) {
            increaseOffset (ename_shipTo);
            res.addChild(parseUSAddress("shipTo"));
      } else throw new InvalidSchema ();

      if (isElement (ename_billTo)) {
            increaseOffset (ename_billTo);
            res.addChild(parseUSAddress("billTo"));
      } else throw new InvalidSchema ();

      if (isElement (ename_comment)) {
            increaseOffset (ename_comment);
            res.addChild(parseString (ename_comment, "comment"));
      }

      if (isElement (ename_items)) {
            increaseOffset (ename_items);
            res.addChild (parseItems ("items"));
      } else throw new InvalidSchema ();

      //tail processing TODO
      if (! EEMoveCursor (ename_purchaseOrder))
            throw new InvalidSchema ();
      return res;
}

private static char[] ename_name = {'n', 'a', 'm', 'e'};
private static char[] ename_street = {'s', 't', 'r', 'e', 'e', 't'};
private static char[] ename_city = {'c', 'i', 't', 'y'};
private static char[] ename_state = {'s', 't', 'a', 't', 'e'};
private static char[] ename_zip = {'z', 'i', 'p'};

Node parseUSAddress (String name) throws InvalidSchema {
      Node res = Node.create ();

      // Parse attributes TODO

      // Parse sequence
      if (isElement (ename_name)) {
            increaseOffset (ename_name);
            res.addChild(parseString (ename_name, "name"));
      } else throw new InvalidSchema ();

      if (isElement (ename_street)) {
            increaseOffset (ename_street);
            res.addChild(parseString (ename_street, "street"));
      } else throw new InvalidSchema ();
```

```
            if (isElement (ename_city)) {
                  increaseOffset (ename_city);
                  res.addChild(parseString (ename_city, "city"));
            } else throw new InvalidSchema ();

            if (isElement (ename_state)) {
                  increaseOffset (ename_state);
                  res.addChild(parseString (ename_state, "state"));
            } else throw new InvalidSchema ();

            if (isElement (ename_zip)) {
                  increaseOffset (ename_zip);
                  res.addChild(parseDecimal (ename_zip, "zip"));
            } else throw new InvalidSchema ();

            //tail processing TODO
            return res;
      }

      private static char[] ename_item = {'i', 't', 'e', 'm'};

      Node parseItems (String name) throws InvalidSchema {
            Node res = Node.create();

            //parse sequence

            while (true) {
                  if (isElement (ename_item)) {
                        increaseOffset (ename_item);
                        res.addChild(parseUnnamed1 ("item"));
                  } else break;
            }

            // tail processing TODO
            return res;
      }

      private static char[] ename_productName = {'p', 'r', 'o', 'd', 'u',
'c', 't',
                  'N', 'a', 'm', 'e'};
      private static char[] ename_quantity = {'q', 'u', 'a', 'n', 't', 'i',
't', 'y'};
      private static char[] ename_USPrice = {'U', 'S', 'P', 'r', 'i', 'c',
'e'};
      private static char[] ename_shipDate = {'s', 'h', 'i', 'p', 'D', 'a',
't', 'e'};

      Node parseUnnamed1 (String name) throws InvalidSchema {
            Node res = Node.create ();

            //processing attribute TODO

            //processing sequence
            if (isElement (ename_productName)) {
                  increaseOffset (ename_productName);
                  res.addChild(parseString (ename_productName,
"productName"));
            } else throw new InvalidSchema ();

            if (isElement (ename_quantity)) {
                  increaseOffset (ename_quantity);
```

```
                res.addChild(parseInteger (1, 99, ename_quantity));
            } else throw new InvalidSchema ();

            if (isElement (ename_USPrice)) {
                increaseOffset (ename_USPrice);
                res.addChild(parseDecimal (ename_USPrice, "USPrice"));
            } else throw new InvalidSchema ();

            if (isElement (ename_comment)) {
                increaseOffset (ename_comment);
                res.addChild(parseString (ename_comment, "comment"));
            }

            if (isElement (ename_shipDate)) {
                increaseOffset (ename_shipDate);
                res.addChild(parseDate (ename_shipDate, "shipDate"));

            }

            //tail processing TODO

            return res;
        }
    }
```

Appendix D

```
/*
 * Created on Aug 17, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package com.docomo.ss;

/**
 * @author zhou
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
public abstract class Parselet {
      public static boolean verify = false;

      protected char[] doc;
      protected int offset;

      public abstract Node parse (char[] doc, int offset);

      protected boolean isElementFrom1 (char[] element) {
            for (int i = 1; i < element.length; i++)
                  if (doc[offset + i] != element[i]) return false;
            return true;
      }

      protected boolean isElement (char[] element) {
            for (int i = 0; i < element.length; i++)
                  if (doc[offset + i] != element[i]) return false;
            if (doc[offset + element.length] == ' ' ||
                        doc[offset + element.length] == '>')
                  return true;
            else return false;
      }

      protected boolean isAttribute (char[] an) {
            for (int i = 0; i < an.length; i++)
                  if (doc[offset + i] != an[i]) return false;
            return true;
      }

      protected Node parseString (char[] name, String ename) throws
InvalidSchema {
            if (doc[offset] != '>') throw new InvalidSchema ();
            int start = ++offset;
            while (doc[offset] != '<' || doc[offset - 1] == '&') offset++;
            Node res = Node.create (ename, new String (doc, start, offset -
1));
            return res;
      }

      protected Node parseDecimal (char[] name, String ename) throws
InvalidSchema {
            System.out.println ("Parselet.parseDecimal not implemented!");
            return null;
      }
```

```
	protected Node parseInteger (int min, int max, char[] name) throws
InvalidSchema {
		System.out.println ("Parselet.parseInteger not implemented!");
		return null;
	}

	protected Node parseDate (char[] name, String ename) throws
InvalidSchema {
		System.out.println ("Parselet.parseDate not implemented!");
		return null;
	}

	protected Node parseDate () throws InvalidSchema {
		System.out.println ("Parselet.parseDate not implemented!");
		return null;
	}

	protected void increaseOffset (char[] ename) {
		offset += ename.length;
	}

	protected void increaseOffset (char[] ename, int additional) {
		offset += ename.length + additional;
	}

	protected boolean EEMoveCursor (char[] ename) {
		if (verify) {
			if (doc[offset] == '<' && doc[offset + 1] == '/') {
				int i;
				for (i = 0; i < ename.length; i++)
					if (doc[offset + i + 2] != ename[i]) return
false;
				if (doc[offset + i + 2] != '>') return false;
			}
		}

		offset += ename.length + 3;
		return true;
	}
}
```

Appendix E

```
/*
 * Created on Aug 17, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package com.docomo.ss;

/**
 * @author zhou
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
public class InvalidSchema extends Exception {
}
```

Appendix F

```
/*
 * Created on Aug 17, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package com.docomo.ss;

/**
 * @author zhou
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
public abstract class Node implements DOMNode {
      public static Node create () {
            System.out.println ("Node.create not implemented!");
            return null;
      }

      public static Node create (String name, Object o) {
            System.out.println ("Node.create not implemented!");
            return null;
      }

      public abstract void addChild (Node n);

      public abstract void addAttribute (Node n, String name);
}
```

Claims

We claim:

1. A method for parsing an XML document, comprising:

Performing an analysis of the XML document from an XML schema associated with the XML document to extract one or more relationships between XML elements included in the XML document; and

parsing the XML elements of the XML document guided by relationships extracted in the analysis.

2. A method as in Claim 1, wherein the relationships extracted from the analysis comprises a lexicographical distance between XML elements.

3. A method as in Claim 1, wherein the relationships extracted from the analysis comprises state-transition information.

4. A method as in Claim 1, further comprising providing element object pools that are created upon system initialization.

5. A method as in Claim 4, further comprising, upon parsing an XML element:

retrieving a corresponding element object from the element object pools; and

filling the element object with values extracted from the parsed XML element.

6. A method as in Claim 5, further comprising providing the element object as a node in a parse tree.

7. A method as in Claim 5, wherein the element object is returned to the element object pools.

8. A method as in Claim 4, wherein each element object in the element object pools correspond to an expected data structure of an XML element defined in the XML schema.

9. A reconfigurable parser for an XML document, comprising:

an analyzer for extracting from an XML schema associated with the XML document relationships between XML elements included in the XML document; and

a parser of the XML elements of the XML document guided by the

relationships extracted by the analyzer.

10. A reconfigurable parser as in Claim 9, wherein the relationships extracted by the analyzer comprises a lexicographical distance between XML elements.

11. A reconfigurable parser as in Claim 9, wherein the relationships extracted by the analyzer comprises state-transition information.

12. A reconfigurable parser as in Claim 9, further comprising element object pools that are created upon system initialization.

13. A reconfigurable parser as in Claim 12, further comprising:

a selector for retrieving a corresponding element object from the element object pools, upon successfully parsing an XML element; and

a writer for filling in the element object with values extracted from the parsed XML element.

14. A reconfigurable parser as in Claim 13, further comprising a parse tree constructor which receives the element object as a node in a parse tree.

15. A reconfigurable parser as in Claim 13, wherein the element object is returned to the element object pools.

16. A reconfigurable parser as in Claim 12, wherein each element object in the element object pools correspond to an expected data structure of an XML element defined in the XML schema.

17. A method for efficiently parsing an XML document, comprising:

analyzing a schema associated with the XML document to extract data structures of XML elements of the XML document;

generating parse code for each data structure of the XML elements; and

parsing the XML elements using the generated parse code as the XML document is read.

18. A method as in Claim 17, wherein the generated parse code is compiled.

19. A method as in Claim 17, further comprising reading the XML elements from the XML document one element at a time.

20. A method as in Claim 19, wherein the XML elements are read into the parser according to a push model.

21. A method as in Claim 19, wherein the XML elements are read into the parser according to a pull model.

22. A method as in Claim 17, further comprising validating the XML elements against the XML schema.

23. A method as in Claim 17, further comprising providing the parsed XML elements in a parse tree.

24. A method as in Claim 17, further comprising providing the parsed XML elements one at a time in a continuous stream.

25. A parser for an XML document, comprising:

a schema analyzer for extracting data structures of XML elements from a schema associated with the XML document;

a parse code generator that generates a parse code for each data structure of the XML elements; and

a parser integrator that invokes a corresponding parse code in respond to each XML element encountered as the XML document is read.

26. A parser as in Claim 25, wherein the generated parse code is compiled.

27. A parser as in Claim 25, further comprising an XML reader that reads the XML elements from the XML document one element at a time.

28. A parser as in Claim 27, wherein the XML elements are read into the parser according to a push model.

29. A parser as in Claim 27, wherein the XML elements are read into the parser according to a pull model.

30. A parser as in Claim 25, wherein the parse code validates the XML elements against the XML schema.

31. A parser as in Claim 25, further comprising an output module that provides the parsed XML elements in a parse tree.

32. A parser as in Claim 25, further comprising an output module that provides the parsed XML elements one at a time in a continuous stream.

5

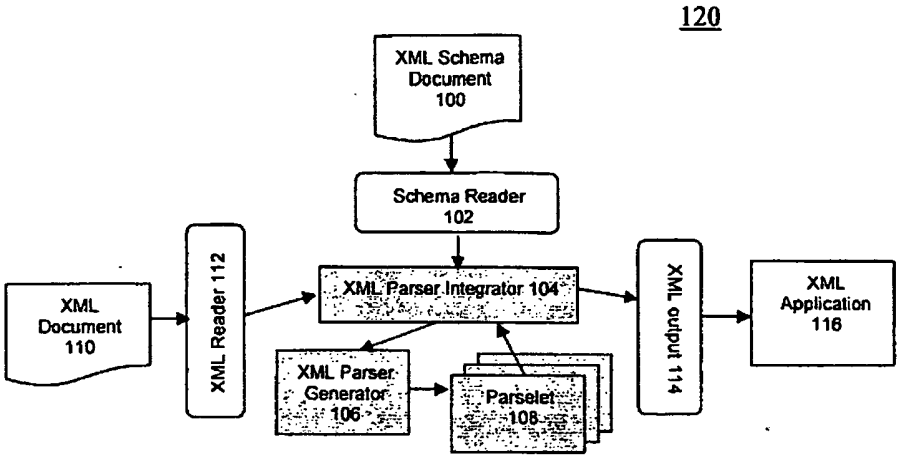

120
XML Schema Document 100
Schema Reader 102
XML Document 110
XML Reader 112
XML Parser Integrator 104
XML output 114
XML Application 116
XML Parser Generator 106
Parselet 108

Figure 1

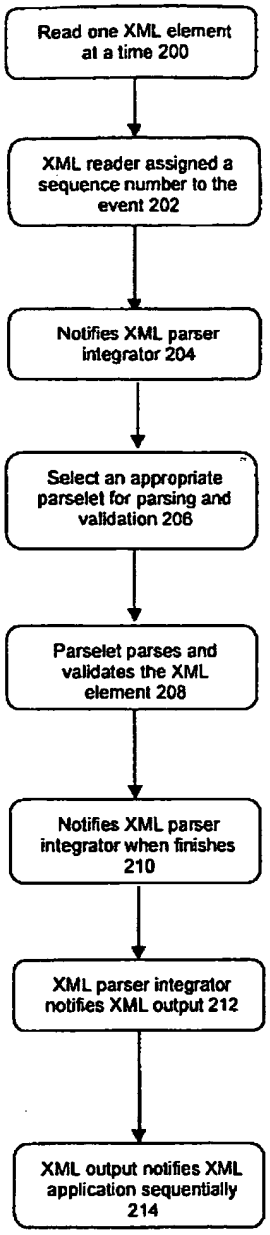

Read one XML element at a time 200
XML reader assigned a sequence number to the event 202
Notifies XML parser integrator 204
Select an appropriate parselet for parsing and validation 206
Parselet parses and validates the XML element 208
Notifies XML parser integrator when finishes 210
XML parser integrator notifies XML output 212
XML output notifies XML application sequentially 214

Figure 2

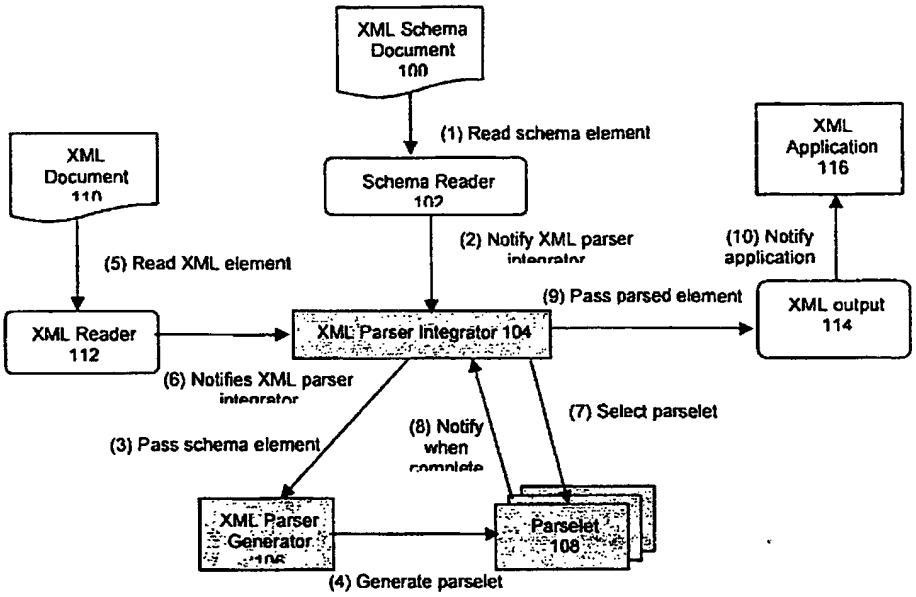

XML Schema Document 100
(1) Read schema element
XML Application 116
XML Document 110
Schema Reader 102
(2) Notify XML parser integrator
(10) Notify application
(5) Read XML element
(9) Pass parsed element
XML output 114
XML Reader 112
XML Parser Integrator 104
(6) Notifies XML parser integrator
(7) Select parselet
(3) Pass schema element
(8) Notify when complete
XML Parser Generator 106
Parselet 108
(4) Generate parselet

Figure 3

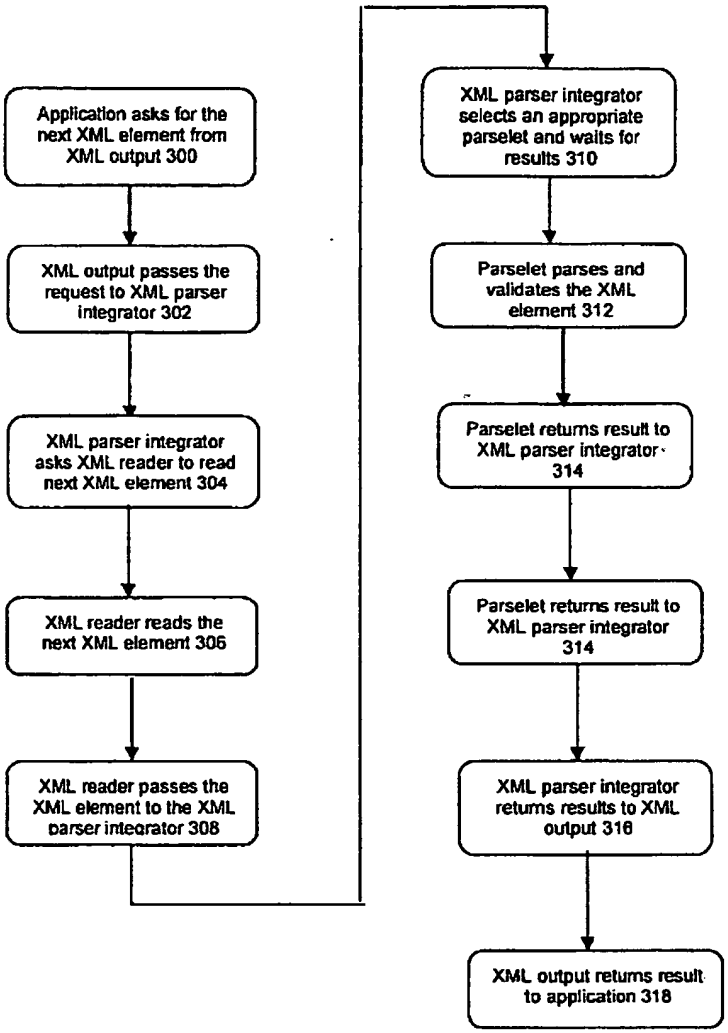

Application asks for the next XML element from XML output 300
XML output passes the request to XML parser integrator 302
XML parser integrator asks XML reader to read next XML element 304
XML reader reads the next XML element 306
XML reader passes the XML element to the XML parser integrator 308
XML parser integrator selects an appropriate parselet and waits for results 310
Parselet parses and validates the XML element 312
Parselet returns result to XML parser integrator 314
Parselet returns result to XML parser integrator 314
XML parser integrator returns results to XML output 316
XML output returns result to application 318

Figure 4

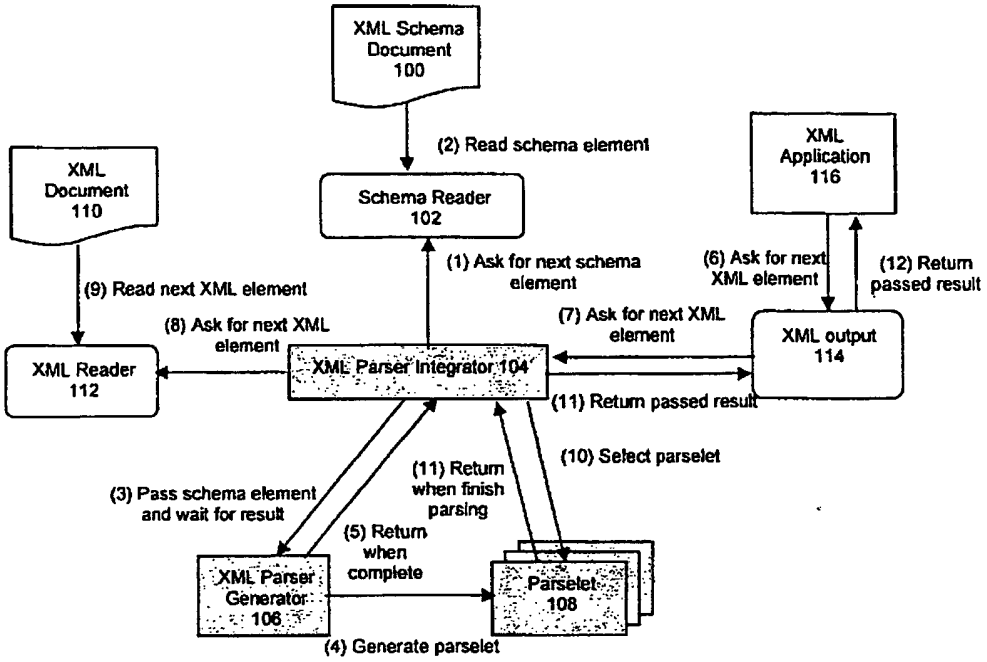

XML Schema Document 100
(2) Read schema element
XML Application 116
XML Document 110
Schema Reader 102
(1) Ask for next schema element
(6) Ask for next XML element
(12) Return passed result
(9) Read next XML element
(8) Ask for next XML element
(7) Ask for next XML element
XML output 114
XML Reader 112
XML Parser Integrator 104
(11) Return passed result
(10) Select parselet
(11) Return when finish parsing
(3) Pass schema element and wait for result
(5) Return when complete
XML Parser Generator 106
Parselet 108
(4) Generate parselet

Figure 5

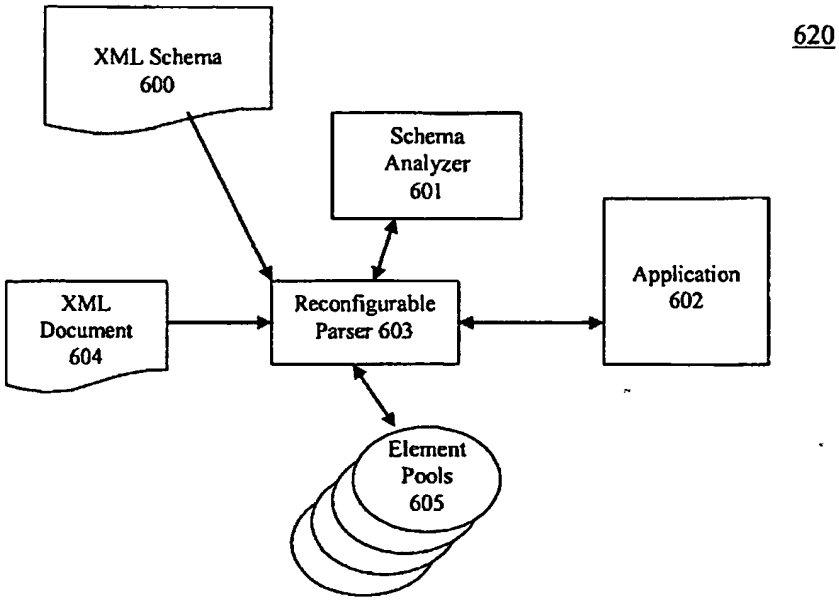

620
XML Schema
600
Schema
Analyzer
601
Application
602
XML
Document
604
Reconfigurable
Parser 603
Element
Pools
605

Figure 6